

AD-A180 065

ADA (TRADENAME) COMPILER VALIDATION SUMMARY REPORT  
RATIONAL ENVIRONMENT (R.E.) INFORMATION SYSTEMS AND  
TECHNOLOGY CENTER W-P AFB OH ADA VALI... 06 MAY 86

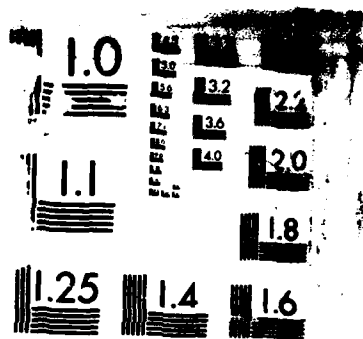
1/1

UNCLASSIFIED

F/G 12/3

NL

END  
100  
100



MM  
No. 1

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

DTIC FILE COPY

2

REPORT DOCUMENTATION PAGE

1. REPORT NUMBER	12. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Ada Compiler Validation Summary Report: Rational Environment A-5-18-1 for the Rational Architecture (R1000)		5. TYPE OF REPORT & PERIOD COVERED 6 MAY 1986 to 6 MAY 1987
7. AUTHOR(s) Wright-Patterson		6. PERFORMING ORG. REPORT NUMBER
9. PERFORMING ORGANIZATION AND ADDRESS Ada Validation Facility ASD/SIOL Wright-Patterson AFB OH 45433-6503		8. CONTRACT OR GRANT NUMBER(s)
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
14. MONITORING AGENCY NAME & ADDRESS (If different from Controlling Office) Wright-Patterson		12. REPORT DATE 6 MAY 1986
		13. NUMBER OF PAGES 40
		15. SECURITY CLASS (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20. If different from Report)

UNCLASSIFIED

DTIC  
ELECTE

MAY 07 1987

D

E

18. SUPPLEMENTARY NOTES

19. KEYWORDS (Continue on reverse side if necessary and identify by block number)

Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

See Attached.

AD-A180 065

DD FORM 1473  
1 JAN 73

EDITION OF 1 NOV 65 IS OBSOLETE  
S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Ada<sup>®</sup> Compiler Validation Summary Report:

Compiler Name: Rational Environment<sup>®</sup> A\_5\_18\_1

Host Computer:  
Rational Architecture (R1000<sup>®</sup>)  
under  
Rational Environment<sup>®</sup>

Target Computer:  
Rational Architecture (R1000)  
under  
Rational Environment

Testing Completed 6 MAY 1986 Using ACVC 1.7

This report has been reviewed and is approved.

Georgeanne G. Chitwood  
Ada Validation Facility  
Georgeanne Chitwood  
ASD/SIOL  
Wright-Patterson AFB, Ohio 45433-6503

John F. Kramer  
Ada Validation Office  
Dr. John F. Kramer  
Institute for Defense Analyses  
Alexandria VA

Virginia L. Castor  
Ada Joint Program Office  
Virginia L. Castor  
Director  
Department of Defense  
Washington DC

Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input checked="" type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By _____	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	



<sup>®</sup>Ada is a registered trademark of the United States Government  
(Ada Joint Program Office).

<sup>®</sup>Rational and R1000 are registered trademarks of Rational.

<sup>®</sup>Rational Environment is a trademark of Rational

87 5 6 127

AVF CONTROL NUMBER: AVF-VSR-28.0786

Ada® COMPILER  
VALIDATION SUMMARY REPORT:  
Rational Environment® A\_5\_18\_1  
for the  
Rational Architecture (R1000®)

Completion of On-Site Validation:  
6 MAY 1986

Prepared By:  
Ada Validation Facility  
ASD/SIOL  
Wright-Patterson AFB, Ohio 45433-6503

Prepared For:  
Ada Joint Program Office  
United States Department of Defense  
Washington, D.C.

---

®Ada is a registered trademark of the United States Government  
(Ada Joint Program Office).

®Rational and R1000 are registered trademarks of Rational.

®Rational Environment is a trademark of Rational

+++++  
+ +  
+ Place NTIS form here +  
+ +  
+++++

- A -

## EXECUTIVE SUMMARY

This Validation Summary Report (VSR) summarizes the results and conclusions of validation testing performed on the Rational Environment® (R1000®), Version A\_5\_18\_1, using Version 1.7 of the Ada® Compiler Validation Capability (ACVC).

The validation process includes submitting a suite of standardized tests (the ACVC) as inputs to an Ada compiler and evaluating the results. The purpose is to ensure conformance of the computer to ANSI/MIL-STD-1815A Ada by testing that it properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent but permitted by ANSI/MIL-STD-1815A. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, or during execution.

On-site testing was performed 5 MAY 1986 through 6 MAY 1986 at Rational, Mountain View, CA under the direction of the Ada Validation Facility (AVF), according to Ada Validation Organization (AVO) policies and procedures. The Rational Environment is hosted on a Rational Architecture (R1000).

The results of validation are summarized in the following table:

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	67	817	1117	14	9	21	2045
Failed	0	0	0	0	0	0	0
Inapplicable	1	7	203	3	2	2	218
Withdrawn	0	4	12	0	0	0	16
TOTAL	68	828	1332	17	11	23	2279

---

®Ada is a registered trademark of the United States Government (Ada Joint Program Office).

®Rational and R1000 are registered trademarks of Rational.

®Rational Environment is a trademark of Rational

There were 16 withdrawn tests in ACVC Version 1.7 at the time of this validation attempt. A list of these tests appears in Appendix D.

Some tests demonstrate that some language features are or are not supported by an implementation. For this implementation, the tests determined the following:

- . SHORT\_INTEGER, SHORT\_FLOAT, and LONG\_FLOAT are not supported.
- . LONG\_INTEGER is supported.
- . Representation specifications for noncontiguous enumeration representations are not supported.
- . Generic unit specifications and bodies can be compiled in separate compilations.
- . Pragma INLINE is not supported for procedures nor for functions.
- . The package SYSTEM is not used by package TEXT\_IO.
- . Modes IN\_FILE and OUT\_FILE are supported for sequential I/O.
- . Instantiation of the package SEQUENTIAL\_IO with unconstrained array types is supported.
- . Instantiation of the package SEQUENTIAL\_IO with unconstrained record types with discriminants is supported.
- . RESET and DELETE are supported for sequential and direct I/O.
- . Modes IN\_FILE, INOUT\_FILE, and OUT\_FILE are supported for direct I/O.
- . Instantiation of package DIRECT\_IO with unconstrained array types and unconstrained types with discriminants without defaults is not supported.
- . Dynamic creation and deletion of files are supported.
- . More than one internal file can be associated with the same external file only for reading.
- . Illegal file names can exist.

ACVC Version 1.7 was taken on-site via magnetic tape to Rational, Mountain View, CA. All tests, except the withdrawn tests and any executable tests that make use of a floating-point precision greater than SYSTEM.MAX DIGITS, were compiled on a Rational Architecture (R1000). Class A, C, D, and E tests were executed on a Rational Architecture.



On completion of testing, execution results for Class A, C, D, or E tests were examined. Compilation results for Class B were analyzed for correct diagnosis of syntax and semantic errors. Compilation and link results of Class L tests were analyzed for correct detection of errors.

The Ada Validation Facility (AVF) identified 2093 of the 2279 tests in Version 1.7 of the ACVC as potentially applicable to the validation of the Rational Environment. Excluded were 170 tests requiring a floating-point precision greater than that supported by the implementation and the 16 withdrawn tests. After the 2093 tests were processed, 48 tests were determined to be inapplicable. The remaining 2045 tests were passed by the compiler.

The AVF concludes that these results demonstrate acceptable conformance to ANSI/MIL-STD-1815A.

## TABLE OF CONTENTS

CHAPTER 1	INTRODUCTION	
1.1	PURPOSE OF THIS VALIDATION SUMMARY REPORT . . . . .	1-1
1.2	USE OF THIS VALIDATION SUMMARY REPORT . . . . .	1-2
1.3	RELATED DOCUMENTS . . . . .	1-3
1.4	DEFINITION OF TERMS . . . . .	1-3
1.5	ACVC TEST CLASSES . . . . .	1-4
CHAPTER 2	CONFIGURATION INFORMATION	
2.1	CONFIGURATION TESTED . . . . .	2-1
2.2	CERTIFICATE INFORMATION . . . . .	2-2
2.3	IMPLEMENTATION CHARACTERISTICS . . . . .	2-3
CHAPTER 3	TEST INFORMATION	
3.1	TEST RESULTS . . . . .	3-1
3.2	SUMMARY OF TEST RESULTS BY CLASS . . . . .	3-1
3.3	SUMMARY OF TEST RESULTS BY CHAPTER . . . . .	3-2
3.4	WITHDRAWN TESTS . . . . .	3-2
3.5	INAPPLICABLE TESTS . . . . .	3-2
3.6	SPLIT TESTS . . . . .	3-4
3.7	ADDITIONAL TESTING INFORMATION . . . . .	3-5
3.7.1	Prevalidation . . . . .	3-5
3.7.2	Test Method . . . . .	3-5
3.7.3	Test Site . . . . .	3-7
APPENDIX A	COMPLIANCE STATEMENT	
APPENDIX B	APPENDIX F OF THE Ada STANDARD	
APPENDIX C	TEST PARAMETERS	
APPENDIX D	WITHDRAWN TESTS	

## CHAPTER 1

### INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard (ANSI/MIL-STD-1815A). Any implementation-dependent features must conform to the requirements of the Ada Standard. The entire Ada Standard must be implemented, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to ANSI/MIL-STD-1815A, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from limitations imposed on a compiler by the operating system and by the hardware. All of the dependencies demonstrated during the process of testing this compiler are given in this report.

VSRs are written according to a standardized format. The reports for several different compilers may, therefore, be easily compared. The information in this report is derived from the test results produced during validation testing. Additional testing information is given in section 3.7 and states problems and details which are unique for a specific compiler. The format of a validation report limits variance between reports, enhances readability of the report, and minimizes the delay between the completion of validation testing and the publication of the report.

#### 1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

- . To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard

## INTRODUCTION

- . To attempt to identify any unsupported language constructs required by the Ada Standard
- . To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by SofTech, Inc., under the direction of the Ada Validation Facility (AVF) according to policies and procedures established by the Ada Validation Organization (AVO). Testing was conducted from 5 MAY 1986 through 6 MAY 1986 at Rational in Mountain View, CA.

### 1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformances to ANSI/MIL-STD-1815A other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse  
Ada Joint Program Office  
OUSDRE  
The Pentagon, Rm 3D-139  
1211 S. Fern, C-107  
Washington DC 20301-3081

or from:

Ada Validation Facility  
ASD/SIOL  
Wright-Patterson AFB, Ohio 45433-6503

Questions regarding this report or the validation tests should be directed to the AVF listed above or to:

Ada Validation Organization  
Institute for Defense Analyses  
1801 North Beauregard  
Alexandria VA 22311

## 1.3 RELATED DOCUMENTS

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, FEB 1983.
2. Ada Validation Organization: Policies and Procedures, MITRE Corporation, JUN 1982, PB 83-110601.
3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., DEC 1984.

## 1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. A set of programs that evaluates the conformance of a compiler to the Ada language specification, ANSI/MIL-STD-1815A.
Ada Standard	ANSI/MIL-STD-1815A, February 1983.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. In the context of this report, the AVF is responsible for conducting compiler validations according to established policies and procedures.
AVO	The Ada Validation Organization. In the context of this report, the AVO is responsible for setting policies and procedures for compiler validations.
Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.
Failed test	A test for which the compiler generates a result that demonstrates nonconformance to the Ada Standard.
Host	The computer on which the compiler resides.
Inapplicable test	A test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.
LMC	The Language Maintenance Committee whose function is to resolve issues concerning the Ada language.
Passed test	A test for which a compiler generates the expected result.
Target	The computer for which a compiler generates code.

## INTRODUCTION

Test	A program that evaluates the conformance of a compiler to a language specification. In the context of this report, the term is used to designate a single ACVC test. The text of a program may be the text of one or more compilations.
Withdrawn test	A test which has been found to be inaccurate in checking conformance to the Ada language specification. A withdrawn test has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

### 1.5 ACVC TEST CLASSES

Conformance to ANSI/MIL-STD-1815A is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Special program units are used to report the results of the Class A, C, D, and E tests at execution. Class B tests are expected to produce compilation errors, and Class L tests are expected to produce link errors.

Class A tests check that legal Ada programs can be successfully compiled and executed. (However, no checks are performed during execution to see if the test objective has been met.) For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a message indicating that it has passed.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntactical or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT-APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no requirements placed on a compiler by the Ada Standard for some parameters (e.g., the number of identifiers permitted in a compilation, the number of units in a library, and the number of nested loops in a subprogram body), a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

## INTRODUCTION

Each Class E test is self-checking and produces a NOT-APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated.

Two library units, the package REPORT and the procedure CHECK\_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report results. It also provides a set of identity functions used to defeat some compiler optimization strategies and force computations to be made by the target computer instead of by the compiler on the host computer. The procedure CHECK\_FILE is used to check the contents of text files written by some of the Class C tests for chapter 14 of the Ada Standard.

The operation of these units is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

Some of the conventions followed in the ACVC are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate tests. However, some tests contain values that require the test to be customized according to implementation-specific values. The values used for this validation are listed in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformance to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The nonconformant tests are given in Appendix D.

## CHAPTER 2

### CONFIGURATION INFORMATION

#### 2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: Rational Environment A\_5\_18\_1

Test Suite: Ada Compiler Validation Capability, Version 1.7

Host Computer:

Machine:	Rational Architecture (R1000)
Operating System:	Rational Environment
Memory Size:	32 Megabytes

Target Computer:

Machine:	Rational Architecture (R1000)
Operating System:	Rational Environment
Memory Size:	32 Megabytes



## CONFIGURATION INFORMATION

### 2.2 CERTIFICATE INFORMATION

#### Base Configuration:

Compiler: Rational Environment A\_5\_18\_1

Test Suite: Ada Compiler Validation Capability, Version 1.7

Certificate Date: 6 MAY 1986

#### Host Computer:

Machine: Rational Architecture (R1000)

Operating System: Rational Environment

#### Target Computer:

Machine: Rational Architecture (R1000)

Operating System: Rational Environment

### 2.3 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. This compiler is characterized by the following interpretations of the Ada Standard:

- . Nongraphic characters.

Nongraphic characters are defined in the ASCII character set but are not permitted in Ada programs, even within character strings. The compiler correctly recognizes these characters as illegal in Ada compilations. The characters are printed in the output listing. One character, form feed (ASCII 12), is not detected because the underlying input system precedes the character with a line terminator which is detected instead. (See test B26005A.)

- . Capacities.

The compiler correctly processes compilations containing loop statements nested to 65 levels and recursive procedures nested to 10 levels. It correctly processes a compilation containing 723 variables in the same declarative part. (See tests D55A03A through D55A03H, D56001B, D64005E through D64005G and D29002K.)

- . Universal integer calculations.

An implementation is allowed to reject universal integer calculations having values that exceed `SYSTEM.MAX_INT`. This implementation does not reject such calculations and processes them correctly. (See tests D4A002A, D4A002B, D4A004A, and D4A004B.)

- . Predefined types.

This implementation supports the predefined type `LONG_INTEGER`. (See tests C34001E, B52004D, B55B09C, C55B07A, and B86001CS.)

- . Based literals.

An implementation is allowed to reject a based literal with a value exceeding `SYSTEM.MAX_INT` during compilation, or it may raise `NUMERIC_ERROR` during execution. This implementation rejects the literal during compilation. (See test E24101A.)

- . Array types.

When an array type is declared with an index range exceeding `INTEGER'LAST` values and with a component that is a null `BOOLEAN` array, this compiler does not raise any exception. (See tests E36202A and E36202B.)

## CONFIGURATION INFORMATION

A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises STORAGE\_ERROR when array objects are declared. (See test C52103X.)

A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises no exception. (See test C52104Y.)

A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC\_ERROR either when declared or assigned. Alternately, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises no exception. (See test E52103Y.)

In assigning one-dimensional array types, the entire expression appears to be evaluated before CONSTRAINT\_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype.

In assigning two-dimensional array types, the entire expression does not appear to be evaluated before CONSTRAINT\_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

### . Discriminated types.

During compilation, an implementation is allowed to either accept or reject an incomplete type with discriminants that is used in an access type definition with a compatible discriminant constraint. This implementation rejects such subtype indications during compilation. (See test E38104A.)

In assigning record types with discriminants, the entire expression appears to be evaluated before CONSTRAINT\_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

### . Aggregates.

In the evaluation of a multi-dimensional aggregate, all choices appear to be evaluated before checking against the index type. (See tests C43207A and C43207B.)

In the evaluation of an aggregate containing subaggregates, all choices are not evaluated before being checked for identical bounds. (See test E43212B.)

All choices are evaluated before CONSTRAINT\_ERROR is raised if a bound in a nonnull range of a nonnull aggregate does not belong to an index subtype. (See test E43211B.)

## CONFIGURATION INFORMATION

### . Functions.

The declaration of a parameterless function with the same profile as an enumeration literal in the same immediate scope is rejected by the implementation. (See test E66001D.)

### . Representation clauses.

'SMALL length clauses are not supported. (See test C87B62C.)

Enumeration representation clauses are not supported. (See test BC1002A.)

### . Pragmas.

The pragma `INLINE` is not supported for procedures nor for functions. (See tests CA3004E and CA3004F.)

### . Input/output.

The package `SEQUENTIAL_IO` can be instantiated with unconstrained array types and record types with discriminants. The package `DIRECT_IO` cannot be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests CE2201D, CE2201E, and CE2401D.)

More than one internal file can be associated with each external file for sequential I/O for reading only. (See tests CE2107A .. F (5 tests (no "E")).)

More than one internal file can be associated with each external file for direct I/O for reading only. (See tests CE2107A .. F (5 tests (no "E")).)

An external file associated with more than one internal file cannot be deleted. (See test CE2110B.)

More than one internal file can be associated with each external file for text I/O for reading only. (See tests CE3111A .. E (5 tests).)

An existing text file can be opened in `OUT_FILE` mode, can be created in `OUT_FILE` mode, and can be created in `IN_FILE` mode. (See test EE3102C.)

Dynamic creation and resetting of a sequential file are allowed. (See test CE2210A.)

Temporary sequential and direct files are given a name. Temporary files given names are deleted when they are closed. (See test CE2108A.)

## CHAPTER 3

### TEST INFORMATION

#### 3.1 TEST RESULTS

The AVF identified 2093 of the 2279 tests in Version 1.7 of the ACVC as potentially applicable to the validation of Rational Environment A\_5\_18\_1. Excluded were 170 tests requiring a floating-point precision greater than that supported by the implementation; and the 16 withdrawn tests. After they were processed, 48 tests were determined inapplicable. The remaining 2045 tests were passed by the compiler.

The AVF concludes that the testing results demonstrate acceptable conformance to the Ada Standard.

#### 3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	67	817	1117	14	9	21	2045
Failed	0	0	0	0	0	0	0
Inapplicable	1	7	203	3	2	2	218
Withdrawn	0	4	12	0	0	0	16
TOTAL	68	828	1332	17	11	23	2279

## TEST INFORMATION

### 3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER												
	2	3	4	5	6	7	8	9	10	11	12	14	TOTAL
Passed	102	232	306	242	160	97	155	197	99	28	216	211	2045
Failed	0	0	0	0	0	0	0	0	0	0	0	0	0
Inapplicable	14	75	88	5	1	0	6	2	6	0	0	21	218
Withdrawn	0	1	4	0	0	0	1	2	6	0	1	1	16
TOTAL	116	308	398	247	161	97	162	201	111	28	217	233	2279

### 3.4 WITHDRAWN TESTS

The following tests have been withdrawn from the ACVC Version 1.7:

B4A010C	C41404A	CA1003B
B83A06B	C48008A	CA3005A through CA3005D (4 tests)
BA2001E	C4A014A	CE2107E
BC3204C	C92005A	
C35904A	C940ACA	

See Appendix D for the test descriptions.

### 3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. For this validation attempt, 218 tests were inapplicable for the reasons indicated:

- . C34001D, C55B07B, B86001CR, B52004E, and B55B09D use SHORT\_INTEGER which is not supported by this compiler.
- . C34001F, C35702A, and B86001CP use SHORT\_FLOAT which is not supported by this compiler.
- . C34001G, C35702B, and B86001CQ use LONG\_FLOAT which is not supported by this compiler.
- . C48006B makes use of a construct whose meaning is still under discussion by the Language Maintenance Committee--the constraining of a discriminated, incomplete type before the elaboration of the full record type declaration. The Rational Environment rejects

## TEST INFORMATION

this test during compilation. Because the issue could not be resolved by the time of validation testing, this test is considered inapplicable.

- . C55B16A makes use of an enumeration representation clause containing noncontiguous values which is not supported by this compiler.
- . B86001DT requires a predefined numeric type other than those defined by the Ada language in package STANDARD. There is no such type for this implementation.
- . C87B62B and C87B62C use 'STORAGE\_SIZE and 'SMALL clauses which are not supported by this compiler.
- . CA3004E, EA3004C, and LA3004A use pragma INLINE for procedures which is not supported by this compiler.
- . CA3004F, EA3004D, and LA3004B use pragma INLINE for functions which is not supported by this compiler.
- . AE2101H uses instantiation of package DIRECT\_IO with unconstrained array types which is not supported by this compiler.
- . CE2102D through CE2102J (6 tests (no "H")) are not applicable since this implementation supports INOUT\_FILE for direct I/O, and IN\_FILE, OUT\_FILE, RESET, and DELETE for sequential and direct I/O.
- . D4A004B requires literals which are outside the range of LONG\_INTEGER. These are not supported by this implementation.
- . D56001B and D64005G require a static nesting level which exceeds the maximum nesting level of 15 supported by this implementation.
- . CE2401B is inapplicable because the creation of sequential I/O and direct I/O files for a type containing access types is not supported by this system.
- . CE2401D is inapplicable because instantiation of package DIRECT\_IO with unconstrained array types and record types with discriminants is not supported by this compiler. However, this test was modified for validation to eliminate instantiation of the package DIRECT\_IO for unconstrained array types and for record types whose discriminants do not have default values. The remaining portions of the test demonstrated that DIRECT\_IO may be instantiated for record types whose discriminants have default values.
- . C92005B is inapplicable because in this system a task's STORAGE\_SIZE attribute yields a value greater than STANDARD.INTEGER'LAST.

## TEST INFORMATION

- . C96005B is inapplicable because in this implementation DURATION is its own base type.
- . CE2107B through CE2107D (3 tests), CE2110B, CE2111D, CE2111H, CE3111B through CE3111E (4 tests), CE3114B, and CE3115A are inapplicable because more than one internal file being associated with the same external file is only supported by this compiler if the files are opened for reading.
- . 170 tests were not processed because they make use of floating-point types requiring more than the maximum precision supported. These tests were:

C24113L through C24113Y (14 tests)  
C35705L through C35705Y (14 tests)  
C35706L through C35706Y (14 tests)  
C35707L through C35707Y (14 tests)  
C35708L through C35708Y (14 tests)  
C35802L through C35802Y (14 tests)  
C45241L through C45241Y (14 tests)  
C45321L through C45321Y (14 tests)  
C45421L through C45421Y (14 tests)  
C45424L through C45424Y (14 tests)  
C45521L through C45521Z (15 tests)  
C45621L through C45621Z (15 tests)

### 3.6 SPLIT TESTS

If one or more errors do not appear to have been detected in a Class B test because of compiler error recovery, then the test is split into a set of smaller tests that contain the undetected errors. These splits are then compiled and examined. The splitting process continues until all errors are detected by the compiler or until there is exactly one error per split. Any Class A, Class C, or Class E test that cannot be compiled and executed because of its size is split into a set of smaller subtests that can be processed.

Splits were required for 40 Class B tests containing more than one syntax error because the compiler stops processing at the first such error.



## TEST INFORMATION

B22003A	B24005B	B32103A	B55A01A
B22004A	B24104A	B35101A	B64001A
B22004B	B24104B	B37201A	B67001A
B22004C	B24104C	B37307B	B67001B
B23004A	B26002A	B41202A	B67001C
B23004B	B26005A	B44001A	B67001D
B24001A	B29001A	B45205A	B97101E
B24001B	B2A003A	B51001A	BB3005A
B24001C	B24003B	B51003A	BC3003A
B24005A	B24003C	B53003A	BC3013A

### 3.7 ADDITIONAL TESTING INFORMATION

#### 3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.7 produced by Rational Environment A\_5\_18\_1 was submitted to the AVF by the applicant for prevalidation review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests.

#### 3.7.2 Test Method

Testing of Rational Environment using ACVC Version 1.7 was conducted on-site by a validation team. The base configuration consisted of a Rational Architecture (R1000) operating under Rational Environment A\_5\_18\_1.

A test tape containing ACVC Version 1.7 was taken on-site by the validation team. The tape contained all tests applicable to this validation as well as all tests inapplicable to this validation except for any Class C tests that require floating-point precision exceeding the maximum value supported by the implementation. Tests that make use of values that are specific to an implementation were customized before being written to the tape. Tests requiring splits during the prevalidation testing were included in their split form on the test tape.

The contents of the tape were loaded directly onto the Rational Architecture using a special program developed by Rational for that purpose. This program read each test file and placed it into a directory based on the test file name. Directories were organized according to chapter so that the tests for a given chapter were placed into the same directory. Some test files were loaded into special directories, however, because of special requirements for running these tests. One special directory contained a subset of Chapter 14 tests that use both CURRENT\_OUTPUT and STANDARD\_OUTPUT and required a special version of the package REPORT (see below). A second special directory contained tests that have side-effects on the program library during compilation, such as

## TEST INFORMATION

the one that redefines the package SYSTEM.

Once all tests had been loaded to disk, three parallel batch streams were started. The Rational Environment includes both an interactive Ada editing facility and support for incremental change to semantically consistent units. Since the ACVC is structured only for testing batch compilers, Rational constructed a batch facility that invokes components of the interactive and incremental compilation system. Tests were run using this batch facility, and thus indirectly using the Rational Environment compilation system, but no explicit testing of these facilities was attempted.

In the special batch environment created for testing purposes, it was necessary to change package body REPORT so that writing was done to CURRENT\_OUTPUT rather than to STANDARD\_OUTPUT because file STANDARD\_OUTPUT was identified with the user's terminal. (Even for specially constructed batch streams STANDARD\_OUTPUT is associated with the terminal used to start the stream.) The test team verified that this was the only change to that package. However, two versions of REPORT were required because some of the executable tests for Chapter 14 check the use of STANDARD\_OUTPUT and CURRENT\_OUTPUT. These tests were run using the standard ACVC version of the REPORT package with the console designated as STANDARD\_OUTPUT. Execution results for these tests were copied to the printer from the console.

After the test files were loaded to disk, the full set of tests was run on the R1000. Results were printed from the R1000. Tests that were withdrawn from ACVC Version 1.7 were not run.

The compiler was tested using command scripts provided by Rational. These scripts were reviewed by the validation team. The following switches were in effect for testing:

<u>Switch</u>	<u>Setting</u>
Account	""
Auto_Login	False
Closed_Private_Part	False
Parser_Configuration	()
Create_Internal_Links	True
Create_Subprogram_Specs	False
Ignore_Interface_Pragmas	False
Ignore_Minor_Errors	False
Ignore_Unsupported_Rep_Specs	False
Password	""
Remote_Directory	""
Remote_Machine	""
Remote_Root	""
Remote_Type	Rational

## TEST INFORMATION

Require_Internal_Links	False
Send_Port_Enabled	False
Subsystem_Interface	False
Target_Key	R1000
Transfer_Mode	Stream
Transfer_Structure	File
Transfer_Type	ASCII
Username	""

Tests were run in batch mode using a single computer. Test output, compilation listings, and job logs were captured on magnetic tape and archived at AVF. The listings examined on-site by the validation team were also archived.

### 3.7.3 Test Site

The validation team arrived at Rational in Mountain View, CA on 5 MAY 1986 and departed after testing was completed on 6 MAY 1986 .

APPENDIX A  
COMPLIANCE STATEMENT

Rational has submitted the following compliance statement concerning the Rational Environment.

## Compliance Statement

### Base Configuration:

Compiler: Rational Environment A\_5\_18\_1

Test Suite: Ada Compiler Validation Capability, Version 1.7

### Host Computer:

Machine: Rational Architecture (R1000)

Operating System: Rational Environment A\_5\_18\_1

### Target Computer:

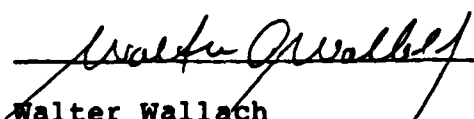
Machine: Rational Architecture (R1000)

Operating System: Rational Environment A\_5\_18\_1

Rational has made no deliberate extensions to the Ada language standard.

Rational agrees to the public disclosure of this report.

Rational agrees to comply with the Ada trademark policy, as defined by the Ada Joint Program Office.

 Date: 5/6/86  
Walter Wallach  
RATIONAL  
Manager, Software Test and Release

## APPENDIX B

### APPENDIX F OF THE Ada STANDARD

The *Ada Language Reference Manual* specifies that certain features of the language are implementation dependent. It requires that these implementation dependencies be defined in an appendix, Appendix F.

This section of the *Rational Environment Reference Manual* is the Appendix F for the Rational Environment and the Rational Architecture. It contains sections that describe the following implementation-dependent features:

- Compilation
- The predefined language environment
- Attributes
- Pragmas
- Representation clauses
- Chapter 14, I/O
- Limits

Note that there will be a separate Appendix F for each additional target supported by the Environment.

### Compilation

The following sections introduce some of the concepts that underlie the Environment compilation system and provide a summary of the separate compilation rules for Ada units in the Environment.

### Unit States

The Rational Environment provides an integrated representation of programs, independent of their compilation state. In the Environment, no distinction is made between source, object code, or other implementation-dependent representations.

In the Environment, each Ada unit can be in one of three basic states that range from source, the lowest state, to coded, the highest state. The process of transforming a program to the state in which it can be executed consists of promoting all of its units from the source state to the coded state and then promoting a command that references it. Each of the states is described in more detail below.

## APPENDIX F OF THE Ada STANDARD

- **Source** - In this state, the image of the unit can be edited. Other units that reference it (in the Ada sense) cannot be in a state higher than the source state.
- **Installed** - In this state, the unit has been syntactically and semantically checked according to the definition of the Ada language. Other units can now reference it (in the Ada sense); that is, they can be promoted from the source state to higher states.
- **Coded** - In this state, code has been generated for the unit, and the unit can be executed from a Command window.

### **Treatment of Generics**

Since the Rational Environment and the Rational Architecture do not depend on macro expansion approaches to compile generics, the specification and the body of a generic are not required to be in the same compilation. Bodies of generics can be changed without obsolescing instantiations of these generics.

If the formal part of a generic contains private (or limited private) types, certain additional implicit dependencies between the specification, body, and instantiations of a generic may be introduced (see Section 13.3.2 of the *Ada Language Reference Manual*). The effect of these implicit dependencies is described more fully in "Installation," below, and in the discussion of the `Must_Be_Constrained` pragma in "Pragmas," later in this section.

### **Installation**

Installation ordering rules follow Ada's separate compilation rules. Specs must be installed before their corresponding bodies are installed. Subunits must be installed after their parents are installed. A unit spec must be installed before another unit that refers to it can be installed. Bodies can be changed without obsolescing using occurrences.

If the formal part of a generic contains private (or limited private) types, certain additional implicit installation dependencies among the specification, body, and instantiations of a generic may be introduced (see Section 13.3.2 of the *Ada Language Reference Manual*).

If the specification and body of such a generic are installed, and the body contains language constructs that would require constrained actuals for the formal private (or limited private) types, instantiations that do not provide constrained actuals for these formals cannot be installed after this point (semantic errors will be generated). If, on the other hand, the specification for such a generic and at least one instantiation with unconstrained actuals for the formals have been installed, the body for the generic cannot then be installed if it contains language constructs that would require constrained actuals (semantic errors will be generated).

The Environment supports a pragma, the `Must_Be_Constrained` pragma, that can be used to provide more explicit control over the treatment of generics with formals that are private (or limited private). More information is available in the description of the `Must_Be_Constrained` pragma in "Pragmas," later in this section.

It is always legal for a generic actual parameter to be a type with discriminants if the discriminants have default values. In generic unit instantiation, the Rational Environment treats such actual parameters as if they were constrained types. This conforms to the requirements of AI-00037.

Literal declarations outside the bounds of `Long_Integer` are rejected at installation time. The bounds of `Long_Integer` are `System.Min_Int..System.Max_Int`.

A parameterless function having the same name and type as the enumeration literal (declared in the same scope) is rejected at installation time. This conforms to AI-00330.

### **Coding**

Specs must be coded before their corresponding bodies are coded. Specs must also be coded before users of those specs can be coded. Changes that require recoding the specs require recoding the bodies of the units that have been changed and all usages of those changed units.

### **Incremental Installing/Withdrawing**

The Rational Environment currently allows certain incremental changes to units in the installed state (note that coded units must be demoted to the installed state before these operations are allowed).

New declarations that are upward compatible (based on Ada semantics) can be inserted into installed units. Entire declarations can be incrementally demoted from installed to source, edited, and then incrementally reinstalled.

The Environment also allows incremental operations on entire statements for units in the installed state. New statements can be inserted or deleted. Existing statements can be incrementally demoted to source, edited, and then reinstalled.

Incremental operations are also allowed in context clauses.

Incremental operations are not allowed for two-part types or generic formal parts.

## **The Predefined Language Environment**

The following material describes the predefined library units: package `Standard`, package `System`, the `Unchecked_Deallocation` procedure, and the `Unchecked_Conversion` function.

### **Package Standard**

Package `Standard` defines all of the predefined identifiers in the language.

package `Standard` is

```
type Boolean is (False, True);
for Boolean'Size use 1;
```



## APPENDIX F OF THE Ada STANDARD

```

type Integer      is range -2**31 .. 2**31;
type Long_Integer is range (-2**62 - 2**62) .. (2**62 - 1 + 2**62);
--                -2**63                .. 2**63-1

type Float is digits 15
           range -1.7977E308 .. 1.7977E308;

type Character is (Nul, ..., Del);
for Character use (0, ..., 127);
for Character'Size use 8;

package Ascii is ... end Ascii;

subtype Natural is Integer range 0 .. Integer'Last;
subtype Positive is Integer range 1 .. Integer'Last;

type String is array (Positive range <>) of Character;

type Duration is delta 3.051757812500E-05
                    range -4.294967296000E+09 .. 4.294967296000E+09;

Constraint_Error : exception;
Numeric_Error    : exception;
Program_Error    : exception;
Storage_Error    : exception;
Tasking_Error    : exception;

end Standard;

STANDARD.DURATION'SMALL = 3.0517578125 E-05 seconds

```

### Package System

Package System defines various implementation-dependent types, objects, and subprograms.

Other declarations are defined in package System that are reserved for internal use and are not documented. These declarations should not be required for users of the Rational Environment.

package System is

```

type Name      is (R1000);

System_Name    : constant Name := R1000;

Bit            : constant := 1;
Storage_Unit   : constant := 1 * Bit;

Word_Size     : constant := 128 * Bit;
Byte_Size     : constant := 8 * Bit;
Megabyte      : constant := (2 ** 20) * Byte_Size;
Memory_Size   : constant := 32 * Megabyte;

-- System-Dependent Named Numbers

Min_Int       : constant := Long_Integer'Pos (Long_Integer'First);
Max_Int       : constant := Long_Integer'Pos (Long_Integer'Last);

```

```

Max_Digits   : constant := 15;
Max_Mantissa : constant := 63;
Fine_Delta   : constant := 1.0 / (2.0 ** 63);
Tick         : constant := 200.0E-9;

```

```

subtype Priority is Integer range 0 .. 5;

```

```

type Byte is new Natural range 0 .. 255;

```

```

type Byte_String is array (Natural range 0) of Byte;
-- Basic units of transmission/reception to/from IO devices.

```

```

Instruction_Error : exception;
-- Raised by Unchecked_Conversion when conversion fails.

```

```

end System;

```

### Unchecked\_Deallocation Procedure

The Unchecked\_Deallocation procedure is used to perform unchecked storage deallocation for objects designated by values of access types other than task types.

```

generic
  type Object is limited private;
  type Name is access Object;
  procedure Unchecked_Deallocation(X : in out Name);

```

Note that the current implementation of Unchecked\_Deallocation assigns null to X but does not actually reclaim the storage for the object it designates. The procedure is provided so that software that eventually will be targeted to other computers can be developed and debugged in the Rational Environment.

### Unchecked\_Conversion Function

The Unchecked\_Conversion function converts objects of one type to objects of another type.

```

generic
  type Source is limited private;
  type Target is limited private;
  function Unchecked_Conversion (S : Source) return Target;

```

This function returns the bit pattern for S as an object of the Target type. If the conversion is impossible, the Instruction\_Error exception is raised.

Unchecked\_Conversion succeeds if the following conditions are met:

- If the Target type is an array, it must be constrained.
- Target'Size must be less than or equal to Source'Size. If Target'Size is smaller than Source'Size, truncation occurs.
- The Source and Target cannot contain access, task, discriminated, or private types.

Examples of conversions that succeed include:

- Long\_Integer to a record with two Integer fields.
- Float to a Long\_Integer.
- Integer and an array of 32 Booleans.
- Two record types of equal size, each containing only scalar subcomponents and no discriminants.

#### **Package Machine\_Code**

The package Machine\_Code is not currently supported.

#### **Attributes**

The Environment supports no implementation-dependent attributes other than those defined in Appendix A of the *Ada Language Reference Manual*. The following clarifications and restrictions complement the descriptions provided in Appendix A:

- 'Address - This attribute is not supported; any number returned is meaningless.
- 'First\_Bit - This attribute is not supported.
- 'Last\_Bit - This attribute is not supported.
- 'Position - This attribute is not supported.
- 'Storage\_Size - 'Storage\_Size is meaningful only when applied to access types or subtypes, in which case it returns the number of storage units reserved for the collection associated with the base type for the access type or subtype. 'Storage\_Size has no effect for task types or task objects.

#### **Pragmas**

The Environment supports pragmas for application software development in addition to those defined in Appendix B of the *Ada Language Reference Manual*. They are described below, along with additional clarifications and restrictions for the pragmas defined in Appendix B of the *Ada Language Reference Manual*:

- Closed\_Private\_Part - This pragma is used in conjunction with the Subsystem tools to indicate that a Subsystem interface has a closed private part.
- Controlled - Since the implementation does not support automatic garbage collection, this pragma is always implicitly in effect.
- Inline - This pragma currently has no effect.
- Interface - The Environment does not currently support the execution of other languages on the Rational Architecture. However, to support development of target-dependent software containing this pragma, the Environment recognises this pragma. The effect of this pragma is that a body is implicitly built that will raise the Program\_Error exception if the subprogram is executed.
- List - This pragma currently has no effect.

- **Main** - This pragma is used to cause the Environment to preload the object code for the compilation units referenced by a main program. Normally this loading is done when a Command window referencing these units is promoted.

The pragma takes no parameters and should be placed immediately after the declaration for the specification or the body of the main subprogram. Note that there is a restriction that the parameters to subprograms containing this pragma must be of types defined in package Standard, package System, or any other predefined package in the Environment directory structure provided by Rational.

The pragma can be placed only after library units. The loading takes place when the body of the main program is promoted to the coded state. In order for this to occur, all compilation units referenced by the main program must be in the coded state.

When using the Subsystem tools, the loading of subprograms containing a Main pragma will use the current Activity to determine the actual Subsystem implementations that will comprise the main program. Once the loading has taken place, the execution of the main program can occur without requiring an Activity.

Executing a main program containing this pragma first causes the closure of the library units referenced by the main program to be elaborated. The program is then executed. If there are references in the Command window to units in the closure of the main program other than within the main program, these references will cause their own copy of these units to be elaborated. These elaborated instances will be separate from those of the main program's elaboration.

- **Memory\_Size** - This pragma has no effect.
- **Must\_Be\_Constrained** - This pragma is used in a generic formal part to indicate that formal private (and limited private) types must be constrained or need not be constrained.

This pragma allows programmers to declare explicitly how they intend to use the formals in the specification for a generic. By doing so, the Environment can check that any instantiations of the generic that are installed before the body of the generic is installed are legal.

Its syntax is: `pragma Must_Be_Constrained ([<cond> =>] <type_id>, ...);`

The <cond> can be either yes or no and defaults to the previous value (which is initially yes) if omitted. <type\_id> must be a formal private (or limited private) type defined in the same formal part as the pragma.

If the <cond> value of no is specified, any use in the body that requires a constrained type will be flagged as a semantic error. If yes is specified, any instantiations that contain actuals that require constrained types will be flagged with semantic errors if the actuals are not constrained.

- **Open\_Private\_Part** - This pragma is used in conjunction with the Subsystem tools to indicate that a Subsystem interface has an open private part.
- **Optimise** - This pragma currently has no effect.
- **Pack** - All records and arrays are stored packed in the minimum number of bits that they require, unless explicitly overridden by a length representation clause (see "Representation of Objects," below). Thus, this pragma has no effect.
- **Page** - This pragma currently has no effect.

- **Priority** - Priorities can be specified only inside a task or a library main program. If multiple priorities are specified, only the first priority specified is used. The default priority is 2.
- **Private\_Eyes\_Only** - This pragma is used in conjunction with the Subsystem tools to indicate that items in a context clause are required only in the private part of a Subsystem interface.
- **Shared** - This pragma currently has no effect.
- **Storage\_Unit** - The only legal storage unit value for the Rational Architecture is 1.
- **Suppress** - This pragma currently has no effect.
- **System\_Name** - The only legal system name is R1000.

## Representation Clauses

The Rational Environment does not currently provide a complete implementation for representation specifications. However, to facilitate host/target development of target-dependent code containing representation clauses, the Environment will optionally ignore unsupported representation clauses.

## Representation of Objects

The Environment follows some simple rules for representing objects in virtual memory, and these rules can be used to create objects with arbitrary bit images without using representation clauses.

For discrete types as components of structures (records and arrays), the Rational-Architecture representation will allocate the minimum amount of space to represent the range imposed by the (possibly dynamic) constraints of the applicable subtype, using a two's complement representation that is zero based.

For example:

```

subtype Binary is Integer range 0 .. 1;    -- uses 1 bit
subtype A is Integer range -3 .. 120;      -- uses 8 bits
type B is new Natural range 0..63;         -- uses 6 bits
type C is new Natural range 1022..1023;    -- uses 10 bits
type D is (X, Y, Z);                       -- uses 2 bits
                                         -- X => 0
                                         -- Y => 1
                                         -- Z => 2
type E is (X);                             -- uses 0 bits
    
```

Size representation clauses are supported for all enumeration types, as long as they are not declared with two-part declarations. Thus, the above rules can be overridden. A specific example of this is the representation for the Standard.Character type, which takes 8 bits instead of 7 because of a size representation clause.

For records without discriminants, the Rational Architecture stores the fields in the order specified in the type declaration, using the minimum space required for each field, with no additional Environment-generated fields.

```

type R1 is                      -- Uses 8+6+1 = 15 bits
  record
    Field_1 : A;
    Field_2 : B;
    Field_3 : Boolean;
  end record;

```

```

type R2 is                      -- Uses 15+1 = 16 bits
  record
    Field_1 : R1;
    Field_2 : Boolean;
  end record;

```

For constrained array types, the Rational Architecture stores the elements packed, using the minimum space for each element, with no additional fields.

```

type A1 is array (1..N) of R1;  -- Uses 15*N bits
                                -- N need not be static

```

```

type A2 is array (0..10) of Boolean; -- Uses 11 bits.

```

```

type R3 is                      -- Uses 15+11+2
  record                        --       = 28 bits
    Field_1 : R1;
    Field_2 : A2;
    Field_3 : D;
  end record;

```

### Length Clauses

- **'Size** - The Rational Architecture supports the **'Size** attribute for discrete types only. These types are further limited in that they can have only a single declaration point (that is, they cannot be incomplete or private types). The size specified must be less than or equal to 64.
- **'Storage\_Size** for collections - The default collection size is  $2^{24}$  bits. The storage size for a collection can range from  $2^8$  to  $2^{32}$  bits. The storage size for a collection determines the number of bits required to represent access types for the collection (for example, for collections of the default  $2^{24}$  bit size, the number of bits required to store objects of the access type that is associated with this collection is 24). Only types with single declaration points can have storage size specified (that is, they cannot be incomplete or private types).
- **'Storage\_Size** for tasks - Since each task in the Rational Architecture gets its own virtual address space, storage size specifications for tasks are meaningless and, consequently, are not supported.
- **'Small** - This length clause is not currently supported.

### Enumeration Representation Clauses

No enumeration representation clauses are currently supported.

### **Record Representation Clauses**

No record representation clauses are currently supported.

### **Address Clauses**

No address clauses are currently supported.

### **Interrupts**

Since interrupts do not exist in the Rational Architecture, these representation clauses are not needed and, consequently, are not supported.

## **Chapter 14 I/O**

The Environment supports all of the I/O packages defined in Chapter 14 of the *Ada Language Reference Manual*, except for package `Low_Level_Io`, which is not needed. The Environment also provides a number of other I/O packages. The packages defined in Chapter 14, as well as the other I/O packages supported by the Environment, are more fully documented in the "Input/Output" section of the *Rational Environment Reference Manual*.

The following list summarises the implementation-dependent features of the Chapter 14 I/O packages:

- **Filenames** - Filenames must conform to the syntax of Ada identifiers. They can, however, be keywords of the Ada language.
- **Form parameter** - The Form parameter is ignored.
- **Instantiations of package `Direct_Io` and package `Sequential_Io` with access types** - Such instantiations are allowed. If files are created or opened using such instantiations, the `Use_Error` exception is raised.
- **Count type** - The Count type for package `Text_Io` and package `Direct_Io` is defined as:

```
package Text_Io is
  type Count is range 0 .. 1_000_000_000;
end Text_Io;

package Direct_Io is
  type Count is new Integer
    range 0 .. Integer'Last/Element_Type'Size;
end Direct_Io;
```

- **Field subtype** - The subtype `Field` for package `Text_Io` is defined as:  

```
subtype Field is Integer range 0 .. Integer'Last;
```
- **Standard\_Input and Standard\_Output files** - These files are the interactive input/output windows provided by the Rational Editor.
- **Internal and external files** - More than one internal file may be associated with a single external file for input only. Only one internal file may be associated with a single external file for output or inout.

- **Sequential\_Io and Direct\_Io packages** - Package **Sequential\_Io** may be instantiated for unconstrained array types or for types with discriminants without default discriminant values. Package **Direct\_Io** may not be instantiated for unconstrained array types or for types with discriminants without default discriminant values.
- **Terminators** - The control characters used as terminators are:  
 Line terminator - **Ascii.Lf**  
 Page terminator - **Ascii.Ff**  
 File terminator - **Ascii.Eot**
- **Treatment of control character** - Control characters, other than the terminators described above, are passed directly to and from files to application programs.
- **Concurrent properties** - The Chapter 14 I/O packages assume that concurrent requests for I/O resources will be synchronised by the application program making the requests.

### Limits

The following package specifies the absolute limits on the use of certain language features:

package **Limits** is

```

  Large : constant := <some very large number>;

  -- Scanner
  Max_Line_Length          : constant := 254;

  -- Semantics
  Max_Discriminants_In_Constraint : constant := 256;
  Max_Associations_In_Record_Aggregate : constant := 256;
  Max_Fields_In_Record_Aggregate : constant := 256;
  Max_Formals_In_Generic : constant := 256;
  Max_Nested_Contexts : constant := 250;
  Max_Nested_Packages : constant := Large;
  Max_Units_In_Transitive_Closure_Of_With_Lists : constant := Large;
  -- (limited by virtual memory stack size)
  Max_Number_Of_Libraries : constant := Large;

  -- Code Generator
  Max_Non_Null_Case_Alternatives : constant := 255;
  Max_Indices_In_Array_Aggregate : constant := 64;
  Max_Parameters_In_Call : constant := 255;
  Max_Expression_Nesting_Depth : constant := Large;
  -- (limited by virtual memory stack size)
  Max_Number_Of_Enumeration_Literals : constant := 256;
  Max_Number_Of_Fields_In_Records : constant := 255;
  Max_Number_Of_Entries_In_A_Task : constant := 255;
  Max_Number_Of_Dimensions_In_An_Array : constant := 63;
  Max_Nesting_Of_Subprograms_Or_Blocks_In_A_Package : constant := 14;

  -- Execution
  Max_Number_Of_Tasks : constant := Large;
  -- (limited by available disk space)

```

end **Limits**;



# APPENDIX C TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are identified by names that begin with a dollar sign. A value is substituted for each of these names before the test is run. The values used for this validation are given below.

<u>Name and Meaning</u>	<u>Value</u>
\$BIG_ID1 Identifier of size MAX_IN_LEN with varying last character.	(1 .. 253 => 'A', 254 => '1')
\$BIG_ID2 Identifier of size MAX_IN_LEN with varying last character.	(1 .. 253 => 'A', 254 => '2')
\$BIG_ID3 Identifier of size MAX_IN_LEN with varying middle character.	(1 .. 126 => 'A', 127 => '3', 128 .. 254 => 'A')
\$BIG_ID4 Identifier of size MAX_IN_LEN with varying middle character.	(1 .. 126 => 'A', 127 => '4', 128 .. 254 => 'A')
\$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is MAX_IN_LEN characters long.	(1 .. 251 => '0', 252 .. 254 => '298')

# TEST PARAMETERS

<u>Name and Meaning</u>	<u>Value</u>
<b>\$BIG_REAL_LIT</b> A real literal that can be either of floating- or fixed-point type, has value 690.0, and has enough leading zeroes to be MAX_IN_LEN characters long.	(1 .. 249 => '0', 250 .. 254 => '69.0E1')
<b>\$BLANKS</b> Blanks of length MAX_IN_LEN - 20	(1 .. 234 => ' ')
<b>\$COUNT_LAST</b> Value of COUNT'LAST in TEXT_IO package.	1_000_000_000
<b>\$EXTENDED_ASCII_CHARS</b> A string literal containing all the ASCII characters with printable graphics that are not in the basic 55 Ada character set.	abcdefghijklmnopqrstuvwxyz!\$%?@[\]^`{}~
<b>\$FIELD_LAST</b> Value of FIELD'LAST in TEXT_IO package.	2147483647
<b>\$FILE_NAME_WITH_BAD_CHARS</b> An illegal external file name that either contains invalid characters or is too long.	BAD_CHARACTERS&<>=
<b>\$FILE_NAME_WITH_WILD_CARD_CHAR</b> An external file name that either contains a wild card character or is too long.	WILDCARDS@
<b>\$GREATER_THAN_DURATION</b> A universal real value that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	5.0E09
<b>\$GREATER_THAN_DURATION_BASE_LAST</b> The universal real value that is greater than DURATION'BASE'LAST.	5.0E09
<b>\$ILLEGAL_EXTERNAL_FILE_NAME1</b> Illegal external file name.	BAD_CHARACTERS&<>=
<b>\$ILLEGAL_EXTERNAL_FILE_NAME2</b> Illegal external file names.	(1 .. 100 => 'A') & (1 .. 100 => 'A') & (1 .. 100 => 'A')

## TEST PARAMETERS

<u>Name and Meaning</u>	<u>Value</u>
<b>\$INTEGER_FIRST</b> The universal integer literal expression whose value is INTEGER'FIRST.	-2147483647
<b>\$INTEGER_LAST</b> The universal integer literal expression whose value is INTEGER'LAST.	2147483647
<b>\$LESS_THAN_DURATION</b> A universal real value that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-5.0E09
<b>\$LESS_THAN_DURATION_BASE_FIRST</b> The universal real value that is less than DURATION'BASE'FIRST.	-5.0E09
<b>\$MAX_DIGITS</b> Maximum digits supported for floating-point types.	15
<b>\$MAX_IN_LEN</b> Maximum input line length permitted by the implementation.	254
<b>\$NEG_BASED_INT</b> A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#fffffffffffffe#
<b>\$NON_ASCII_CHAR_TYPE</b> An enumerated type definition for a character type whose literals are the identifier NON_NULL and all non-ASCII characters with printable graphics.	non-null

## APPENDIX D

### WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. When testing was performed, the following 16 tests had been withdrawn at the time of validation testing for the reasons indicated:

- . B4A010C: The object declaration in line 18 follows a subprogram body of the same declarative part.
- . B83A06B: The Ada Standard 8.3(17) and AI-00330 permit the label LAB\_ENUMERAL of line 80 to be considered a homograph of the enumeration literal in line 25.
- . BA2001E: The Ada Standard 10.2(5) states: "Simple names of all subunits that have the same ancestor library unit must be distinct identifiers." This test checks for the above condition when stubs are declared. However, the Ada Standard does not preclude the check being made when the subunit is compiled.
- . BC3204C: The file BC3204C4 should contain the body for BC3204C0 as indicated in line 25 of BC3204C3M.
- . C35904A: The elaboration of subtype declarations SFX3 and SFX4 may raise NUMERIC\_ERROR (instead of CONSTRAINT\_ERROR).
- . C41404A: The values of 'LAST and 'LENGTH are incorrect in IF statements from line 74 to the end of the test.
- . C48008A: This test requires that the evaluation of default initial values not occur when an exception is raised by an allocator. However, the Language Maintenance Committee (LMC) has ruled that such a requirement is incorrect (AI-00397/01).

## WITHDRAWN TESTS

- . C4A014A: The number declarations in lines 19-22 are incorrect because conversions are not static.
- . C92005A: At line 40, "/"=" for type PACK.BIG\_INT is not visible without a USE clause for package PACK.
- . C940ACA: This test assumes that allocated task TT1 will run prior to the main program, and thus assign SPYNUMB the value checked for by the main program; however, such an execution order is not required by the Ada Standard, so the test is erroneous.
- . CA1003B: This test requires all of the legal compilation units of a file containing some illegal units to be compiled and executed. According to AI-00255, such a file may be rejected as a whole.
- . CA3005A..D (4 tests): No valid elaboration order exists for these tests.
- . CE2107E: This test has a variable, TEMP\_HAS\_NAME, that needs to be given an initial value of TRUE.

ATE  
LMED  
-8